Using Streaming SIMD Extensions 2 (SSE2) for SAXPY/DAXPY

Version 2.0

7/00

Order Number: 248600-001

Table of Contents

Revision History

Revision	Revision History	Date
2.0	Pentium® 4 processor update	7/00
1.0	Original publication of document	9/99

References

The following documents are referenced in this application note, and provide background or supporting information for understanding the topics presented in this document.

Lawson, Hanson, Kincaid and Krogh. *"Basic linear algebra subprograms for Fortran usage"*, ACM Transactions on Mathematical Software, Vol. 5, No. 3, page 308-371.

Dongarra, Moler, Bunch, and Stewart, *LINPACK User's Guide,* SIAM, 1979.

Intel Corporation, *C++ Class Libraries for SIMD Operation Reference Manual*, order number 693500, 1999.

Intel Corporation, *Intel® C/C++ Compiler User's Guide*, order number 741901, 1999.

1 Introduction

The Streaming SIMD Extensions 2 (SSE2) introduces new Single Instruction Multiple Data (SIMD) double-precision floating-point instructions and new SIMD integer instructions into the IA-32 Intel® architecture. The double-precision SIMD instructions extend functionality in a manner analogous to the single-precision instructions introduced with the Streaming SIMD Extensions (SSE). The 128-bit SIMD integer extensions are a full superset of the 64-bit integer SIMD instructions, with additional instructions to support more integer data types, conversion between integer and floating-point data types, and efficient operations between the caches and system memory. These instructions provide a means to accelerate operations typical of 3D graphics, real-time physics, spatial (3D) audio, video encoding/decoding, encryption, and scientific application. This application note describes SAXPY and DAXPY, common linear algebra routines in single and double-precision floating-point versions, and includes examples of code that exploit the SSE2 instructions to speed up the implementation of the double-precision version. The single-precision version is identical to previous implementations using SSE instructions, but it still performs better on a Pentium® 4 processor than on a Pentium III processor due to microarchitectural improvements. Since statements about the SAXPY version of the code may also apply to the DAXPY version of the code, this document will adopt the practice of referring to "SAXPY (DAXPY)" when a comment applies to both versions.

SAXPY (DAXPY) computes a constant times a vector plus a vector. It is an elementary vector operation commonly used in linear algebra. The routine is frequently used as a simple benchmark for computer performance, especially within the high performance computing community. This document discusses the coding of SAXPY (DAXPY) exploiting the SIMD instructions available for Intel® Architecture (IA) processors, using both the C++ Class Libraries for SIMD Operations and the vectorization capabilities of the Intel® C/C++ Compiler.

The study in this document is limited to a "perfect cache" environment. That is, data sizes were chosen such that all data was resident in the first level processor cache. Furthermore, all data is assumed to be 16-byte aligned. It is recognized that these restrictions are not realistic for typical code that uses SAXPY (DAXPY). The movement of data between memory and cache has a major impact on performance, and alignment of data cannot be assured *a priori*. Nevertheless, this study serves as a useful demonstration of coding practices that exploit SIMD processor capabilities using the Intel C/C++ Compiler.

2 SAXPY and DAXPY

SAXPY (DAXPY) are defined in BLAS, the Basic Linear Algebra Subprograms, developed by Lawson, Hanson, Kincaid, and Krogh [Lawson, 1979]. SAXPY means single-precision scalar (SA) times the vector X plus the vector Y. DAXPY is the same thing for double-precision data. Expressed mathematically:

```
Y = a * X + Y
```

where `X` and `Y` are vectors (from 1 to n), and `a` is a scalar. The function is usually defined as a FORTRAN subroutine:

```
CALL SAXPY (N, A, X, INCX, Y, INCY)
```

where `N` is the vector length, and `INCX` and `INCY` are used to allow the function to operate on vectors whose elements are not contiguous in memory. However, SAXPY is most often used where the vectors `X` and `Y` are contiguous, so `INCX=INCY=1`. In fact, a common linear systems package known as LINPACK which uses BLAS makes all its calls with `INCX=INCY=1` [Dongarra, 1979].

In the high performance computing community, and in particular, the super-computing community, the performance of LINPACK and BLAS has received much attention, primarily because of the scientific focus of those groups. Performance numbers are frequently quoted for SAXPY and DAXPY. In this document, we will assume that `INCX=INCY=1` to allow the use of SIMD instructions. We will code the functions SAXPY and DAXPY in C++.

2.1 Implementing SAXPY and DAXPY

The most common technique for exploiting the SSE2 instructions is to use hand coded assembly language, usually by writing inline assembly code from within a C/C++ program. However, the Intel C/C++ Compiler provides two alternative choices that offer attractive performance gains with significant ease of use advantages. The C++ Class Libraries for SIMD Operations provide a way to exploit intrinsic versions of the SSE2 instructions in a fairly transparent manner. A more sophisticated approach is to employ the vectorizing capability of the Intel C/C++ Compiler to automatically detect and exploit vector opportunities in the code. These two techniques are described in the following two sections.

2.1.1 Using the C++ Class Libraries for SIMD Operations

A simple way to integrate vector instructions in your application is to use the `FVEC` and `DVEC` classes. The end of this document provides an example. If the data is 16-byte aligned, then you can cast a float (double) pointer to a `F32vec4` (`F64vec2`) pointer, and directly access your data in a natural fashion. Thus, if `x` and `y` are `F32vec4` (`F64vec2`) pointers, you can write `y[I] = scalar * x[I] + y[I]`, and the compiler will generate 128-bit loads and stores, in addition to the appropriate packed arithmetic instructions. You need only adjust your loop count since you are processing 4 (2) elements at a time. If you examine the included code, you will observe that pointers have been cast as described, and loop counts adjusted appropriately. The example code assumes that the vector length, n, is a multiple of 4 (2), so that no 'leftover' elements remain at the end of the vector.

Notice the use of the initializer for the scalar `sa` (`da`). When a single value is used as an initial value, that value is broadcast to all 4 (2) elements of the packed variable `sa` (`da`). All in all, there are very few changes to the original scalar source code. In fact, the core loop is completely unchanged. The ability to use new instructions with such small source code changes yields big improvements in programmer productivity and source code readability.

It is appropriate to add one warning. If the data is not guaranteed to be 16-byte aligned, then casting the float (double) pointer to an `F32vec4` (`F64vec2`) pointer is bad practice. This results in code that works when the data happens to be aligned, but fails when the data is not aligned. If the data is unaligned, but both the `x` and `y` array have the *same* alignment, then it is possible to compute a few unaligned elements in a preamble code, and then treat the rest as aligned. If `x` and `y` have different alignment, then only one array can be accessed as aligned, and a significant performance penalty will result. The Intel C/C++ Compiler supports a simple way to align variables via `declspec` aligns—see the *C/C++ Compiler User's Guide* for information on how to use this feature. If the alignment of `x` and `y` is unknown *a priori,* one can test the alignment of the pointers at run time and select the appropriate version of SAXPY (DAXPY) with an `if` statement.

2.1.2 Using the Vectorizer of the Intel® C/C++ Compiler

Since it is well known that it can be very costly to re-write code, researchers have been working for 30 years to develop compiler technology that automatically "vectorizes" the original code. To apply some of that technology to SAXPY (DAXPY), use the `/QxK /QxW` switches of the Intel C/C++ Compiler.

This turns on the vectorizer, and allows the compiler to generate SSE instructions along with SSE2 instructions. Using the `/Qvec_verbose3` switch gives diagnostic information on what loops can be vectorized, and why other loops cannot be vectorized. These switches allow programmers to rework their code and guide the compiler to do the "heavy-lifting" code transformations. For SAXPY (DAXPY), the Intel C/C++ Compiler produces code with performance comparable to that of hand written assembly code.

When the compiler is presented with a function such as SAXPY (or DAXPY), it often is unable to transform the code because it cannot prove certain conditions. This problem is much worse for C/C++ than it is for FORTRAN due to the language semantics. For instance, the vectorizer may fail to vectorize a function because it will be unable to determine that the pointers passed in actually refer to distinct arrays. With the verbose switch mentioned above, the vectorizer may complain about assumed dependencies between these arrays. While this may seem silly to a programmer, compilers take these things very seriously. To reassure the compiler that this "assumed" dependency is not a concern, the "`ivdep`" pragma can be used. The programmer adds this pragma immediately prior to the `for` loop in question. The loop is then vectorized, with appropriate code inserted to take care of cases where the vector length is not a multiple of 4 (2). However, the compiler still does not know that both the arrays are aligned. The programmer asserts this via the "vector aligned" pragma.

However, the Intel C/C++ Compiler is able to vectorize the SAXPY and DAXPY functions without the use of the "ivdep" and "vector aligned" pragmas. Instead, it inserts code to perform runtime checks for array dependencies and vector alignment. The vectorizing compiler code example provided still includes these pragmas in order to skip the runtime check. On one loop, to allow the sample code to produce baseline marks the C/C++ code example uses the "novector" pragma to prevent compiler vectorization.

One additional important feature of the vectorizer is that the `/QxW` switch can be replaced by the `/QaxW` switch. The first switch allows the compiler to insert SSE2 instructions (and also Pentium III processor instructions). Since that means the code only executes on a Pentium 4 processor, the programmer needs to include a version of the code that does not have SSE2 instructions in order to execute on legacy IA processors. The `/QaxW` switch tells the compiler to include this version automatically. Thus, any function containing SSE2 instruction opportunities is compiled twice—once with SSE2 instructions, as if a `/QxW` switch were present, and once without SSE2 instructions, as if no `/QxW` switch were present. The resultant "fat binary" would run on any Intel processor, delivering high performance on the newer processors, and an equivalent function (at lower performance) on earlier processors. The choice between the two code versions is done at run time, allowing a single binary on a server to be executed in a hybrid environment. This technique produces larger binaries but can provide a simple "code path" solution for many applications.

3 Performance

Performance was measured assuming a "perfect cache" environment. Under this environment, all codes using SSE2 instructions delivered the same performance. That is, the `FVEC` (`DVEC`) code had performance equal to the best hand coded assembly. This is not surprising, given the fact that the C++ Class Libraries for SIMD Operations generated code that was comparable to good quality hand-coded assembly. However, the vectorized code generated by the Intel C/C++ compiler does not perform as well as the `FVEC (DVEC)` code. This is because, at least at the time of this writing, the Intel C/C++ compiler inserts software prefetching into the vectorized code. Because the vector arrays are already in the cache, prefetching can only incur a performance penalty. In general though, the SSE instructions and SSE2 instructions deliver much greater performance than the scalar version of the same code since they perform multiple operations at once (4 for SSE instructions, 2 for SSE2 instructions).

For SAXPY, one would expect SSE instructions to provide a 4x improvement over scalar code. On a Pentium 4 processor a speedup of slightly less than that is achieved, whereas on a Pentium III processor, one only sees improvements on the order of 2x since the memory system is unable to keep pace with the execution unit. This difference is due to improvements in the internal micro-architecture of the processor, which allow the full potential of the SSE instructions to be exploited. For DAXPY, the double precision SSE2 instructions also provide a significant speedup.

4 Conclusion

In summary, to include SIMD instructions such as SSE instructions or SSE2 instructions in a code, there are at least two alternatives to the tedium of hand writing assembly code. These two techniques are: (1) using the C++ Class Libraries for SIMD Operations, which are a wrapper for the intrinsics implemented in the Intel C/C++ Compiler; and (2) using a few pragmas if necessary to exploit the vectorizing capabilities of the Intel C/C++ Compiler. Both techniques will be supported on Intel's IA64 platforms. The latter technique has the additional benefit of greater source code portability. Furthermore, the vectorized code can be invoked within the context of an automatic code path solution, simplifying many code maintenance and distribution issues.

5 C/C++ Coding Example

```
*
 *Saxpy from BLAS, Lawson, Manson, Kincaid, and Krogh (1979)
 *
 *this compilation from _Linpack Users' Guide_, Dongarra, Moler,
 *   Bunch, & Stewart, Siam 1979, appendix A.
 *
 *These versions are "unit" saxpy (daxpy), meaning they assume stride = 1.
 *(note that this is what BLAS requires, and is the most common form)
 *
 * Assume all vectors are aligned.
 *
 */

void usaxpy (int n, float sa, float *sx, float *sy)
{
  if (sa == 0.0) return;

  //The latest intel compilers can now vectorize this loop.
  //Use the novector pragma to prevent vectorization.
  #pragma novector
  for (int i = 0; i < n; i++)
    sy[i] = sa * sx[i]  +  sy[i];
}

void udaxpy (int n, double da, double *dx, double *dy)
{
  if (da == 0.0) return;

  //The latest intel compilers can now vectorize this loop.
  //Use the novector pragma to prevent vectorization.
  #pragma novector
  for (int i = 0; i < n; i++)
    dy[i] = da * dx[i]  +  dy[i];
}
```

6 SSE2 C++ Classes Code Example

```
/* Assumes vectors are aligned, and that the vector length n is divisible
 * by 4 for saxpy and 2 for daxpy.
 */
#include <fvec.h>
#include <dvec.h>

void usaxpy_fvec (int n, float sa, float *sx, float *sy)
{
  F32vec4 *x = (F32vec4 *)sx, *y = (F32vec4 *)sy;
  F32vec4 a(sa);

  if (sa == 0.0) return;
  n >>= 2;
  for (int i = 0; i < n; i++)
    y[i] = a * x[i]  +  y[i];
}
void udaxpy_dvec (int n, double da, double *dx, double *dy)
{
  F64vec2 *x = (F64vec2 *) dx, *y = (F64vec2 *) dy;
  F64vec2 a(da);

  if (da == 0.0) return;
  n >>= 1;
  for (int i = 0; i < n; i++)
    y[i] = a * x[i]  +  y[i];
}
```

7 SSE2 New Vectorizing Compiler Code Example

```
/* Assumes vectors are aligned */

void usaxpy_vec (int n, float sa, float *sx, float *sy)
{
  if (sa == 0.0) return;

  #pragma ivdep
  #pragma vector aligned
  for (int i = 0; i < n; i++)
    sy[i] = sa * sx[i]  +  sy[i];
}

void udaxpy_vec (int n, double da, double *dx, double *dy)
{
  if (da == 0.0) return;

  #pragma ivdep
  #pragma vector aligned
  for (int i = 0; i < n; i++)
    dy[i] = da * dx[i]  +  dy[i];
}
```

Appendix A - Performance Data

Performance Data Revision History

Revision	Revision History	Date
2.0	Updated with 1.2 GHz Pentium 4 performance data	7/00
1.0	Original publication of document	9/99

Table 1: Performance Data for SAXPY/DAXPY Implementations

Performance Data in MFLOPS		
	Pentium III Processor (733 MHz)	Pentium 4 Processor (1.2 GHz)
SAXPY: C code	360	523
SAXPY: FVEC	837	1932
SAXPY: vectorizer	865	1476
DAXPY: C code	353	640
DAXPY: DVEC	N/A	951
DAXPY: vectorizer	N/A	672

Table 1 summarizes the performance results measured assuming a "perfect cache" environment. Vector lengths were chosen to be as long as possible while still fitting into the first level cache. Performance was measured using a 733 MHz Pentium III processor and a 1.2 GHz Pentium 4 processor. See Test Systems Configuration on page 2 for a detailed description of both systems. For each element of the vector, two floating-point operations (one multiply and one add) are performed, and FLOPS rates were computed accordingly.

For SAXPY, one would expect Streaming SIMD Extensions (SSE) to provide a 4x improvement over scalar code. On a Pentium III processor, one only sees improvements on the order of 2x. However on a Pentium 4 processor, a speedup of 3.7x can be achieved. This is due to improvements in the internal micro-architecture of the processor, allowing the full potential of the SSE to be exploited. Notice that a peak speed of 1.9 GFLOPS was achieved.

For DAXPY, the double precision SSE2 provide about a 1.5x speedup. A peak rate of 951 MFLOPS was achieved, which is about half of the single-precision performance. This is because in DAXPY two floating-point operations can occur on a single SIMD instruction; whereas in SAXPY, four floating-point operations can occur.

Test Systems Configuration

Table 2: Pentium III Processor Configuration

Processor	Pentium III Processor at 733 MHz
System	Intel® Desktop Board VC820
Bios Version	VC82010A.86A.0028.P10
Secondary Cache	256KB
Memory Size	128 MB RDRAM PC800-45
Ultra ATA Storage Driver	Production Candidate 6.00.012
Hard Disk	IBM DJNA-371800 ATA-66
Graphics	Creative Labs 3D Blaster† Annihilator† Pro AGP nVidia GeForce256† DDR –32MB
Video Driver Revision	Nvidia Reference Driver 5.22
Operating System	Windows† 2000 Build 2195

Table 3: Pentium 4 Processor Configuration

Processor	Pentium 4 Processor at 1.2 GHz
System	Intel Desktop Board D850GB
Bios Version	GB85010A.86A.0014.D.0007201756
Secondary Cache	256KB
Memory Size	128 MB RDRAM PC800-45
Ultra ATA Storage Driver	Production Candidate 6.00.012
Hard Disk	IBM DJNA-371800 ATA-66
Video Controller/Bus	Creative Labs 3D Blaster Annihilator Pro AGP nVidia GeForce256 DDR –32MB
Video Driver Revision	NVidia Reference Driver 5.22
Operating System	Windows 2000 Build 2195